Enterprise Architektur-Muster

JULIAN BRUDER*, ABDELLAH FILALI*, and LUCA FRANKE*, Hochschule für Technik, Wirtschaft und Kultur Leipzig (HTWK Leipzig), Deutschland

Blah abstrakt…

1 EINLEITUNG

Mit E-Commerce-Beispiel motivieren

2 GRUNDLAGEN VON ENTERPRISE-ARCHITEKTUREN

Verteilte Systeme …
Architekturen …
Komponenten …
…

3 KLASSISCHE ENTERPRISE-ARCHITEKTUREN

Blah …

4 MODERNE ENTERPRISE-ARCHITEKTUREN

4.1 Microservices Architecture

Die Microservices Architecture zeichnet sich grundlegend durch die Unterteilung der Anwendung in einzelne Service Komponenten aus. Jede Service Komponente kann vom Umfang und Granularität variieren und enthält bestenfalls eine einzige isolierte Funktionalität oder einen unabhängigen, größeren Teil der Business Logik. Diese Service Komponenten bestehen aus einen oder mehreren Modulen, die die Funktionalität des Services implementieren.

Wie in Abbildung 1 dargestellt, werden alle von Clients ausgehenden API-Anfragen zunächst von einem API-Gateway entgegengenommen. Dieser kommuniziert dann wiederrum über REST-Anfragen mit den Service-Komponenten, die dann die Anfrage verarbeiten und ein Resultat an das Gateway zurückgeben [Richards 2015, 30].

Ein maßgebendes Merkmal dieser Architektur-Pattern ist die Entkopplung der einzelnen Services voneinander. Dies wird dadurch erreicht, dass alle Services nur über ihre definierten Schnittstellen, miteinander kommunizieren können. Die hohe Entkopplung der Komponenten ermöglicht es, Services unabhängig voneinander zu entwickeln, zu testen und zu deployen. Pipeline- und Deploymentprozesse finden also isoliert voneinander statt, was die Auslieferung der Software beschleunigt [Richards 2015, 27].

Die richtige Granularität der Services ist entscheidend, um die Vorteile der Microservices Architecture voll auszuschöpfen und stellt hier die größte Herausforderung im Design der Architektur dar. Werden die Komponenten zu groß gewählt, so kann es passieren, dass die Vorteile der Microservices Architecture nicht mehr greifen. Innerhalb der Komponenten entstehen Abhängigkeiten, die das isolierte Entwickeln oder Testen von Features verhindern. Auf der anderen Seite führt eine zu kleine Granularität dazu, dass die Anzahl der Services zu groß wird und das Orchestrieren dieser

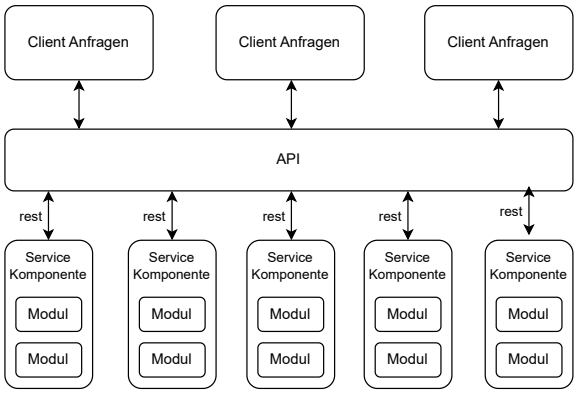

Abb. 1. Aufbau einer Microservices Architecture

einen zu großen Overhead verursacht. Komplexität und Wartungsaufwand steigen stark an und die Schwächen einer klassischen SOA-Architektur treten wieder auf [Richards 2015, 32].

Die Microservices Architektur löst viele der Probleme, die in klassischen monolithischen oder SOA-Architekturen auftreten. Durch die Aufteilung der Anwendung in kleinere, separat deployte Services, entstehen inhärent robustere und besser skalierbare Systeme. Da Änderungen oft nur einzelne Service Komponenten betreffen, muss nicht das ganze System neu deployt werden. Es reicht stattdessen, nur den betroffenen Service zu aktualisieren, was die Ausfallzeiten minimiert und die Wartbarkeit der Software erhöht. Vor allem in agilen Umgebungen, in denen viele Iterationen und schnelle Auslieferungen notwendig sind, ist die Microservices Architektur daher besonders geeignet [Richards 2015, 33].

Auch können Services in Produktiven-Umgebungen ohne Ausfallzeiten ausgetauscht werden. Solange, wie der neue Service deployt wird, leitet das API-Gateway Anfragen an den alten Service weiter. Erst wenn der neue Service bereit ist, werden Anfragen an diesen weitergeleitet und der alte Service kann abgeschaltet werden.

Microservices sind außerdem sehr gut testbar. Tests können durch die strikte Trennung der Funktionalitäten isoliert auf einzelne Services angewendet werden, ohne dass andere Services betrachtet werden müssen. Auch Regressionstests nehmen dadurch weniger Zeit in Anspruch. Kleine Änderungen haben nicht zur Folge, dass die gesamte Anwendung neu getestet werden muss.

Time-to-market bei Microservice Architekturen ist in der Regel kürzer, da deployments schnell sind und parallel an verschiedenen Services gearbeitet werden kann. Es gibt keine zentrale Abhängigkeit, die sich schwerwiegend auf die Entwicklung einzelner Komponenten auswirken kann.

Der größte Nachteil der sich hier ergibt, ist die vergleichsweise niedrige Performance aufgrund der verteilten Natur des Systems.

*Alle Studierenden trugen zu gleichen Teilen zu dieser Arbeit bei.

Diese Arbeit wurde im Rahmen des Mastermoduls „Software Engineering“ (Dozent: Prof. Dr. Andreas Both) an der HTWK Leipzig im Wintersemester 2024/2025 erstellt.

Kommunikation zwischen Services findet meist über das Netzwerk statt, was zu erhöhten Latenzzeiten führen kann[Richards 2015, 34].

4.2 Microkernel Architecture

Die Microkernel Architecture ist ein Architekturmuster, was sich durch Erweiterbarkeit, Flexibilität und vor allem Isolation der Funktionalitäten auszeichnet. Wie in Abbildung 2 dargestellt, enthält ein Microkernel zwei wesentliche Komponenten: Den Kern der Anwendung,der die wichtigsten grundlegenden Funktionalitäten bereitstellt und Module oder auch Plugins, die diesen Kern um Features erweitern [Richards 2015, 21-22].

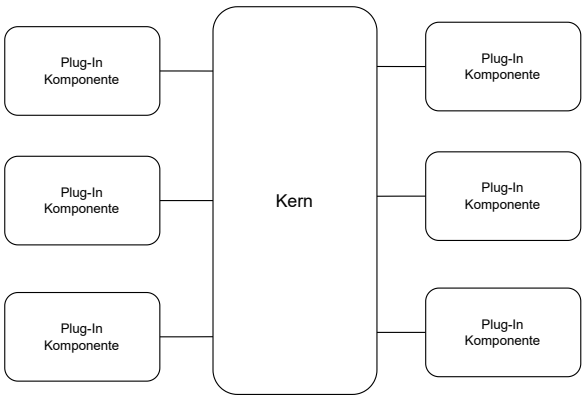

Abb. 2. Aufbau einer Microkernel Architecture

Der Kern der Anwendung implementiert dabei meist nur die minimalste Funktionalität, um die Anwendung oder das System lauffähig zu machen. Alle weiteren Funktionalitäten werden in Modulen implementiert, die auf den Kern aufbauen. Module sind meist unabhängig voneinander aufgebaut, es kann jedoch auch vorkommen, dass manche Module von anderen abhängig sind. Best practice für die Entwicklung von Microkernel Architekturen ist es, die Kommunikation zwischen einzelnen Modulen so gering wie möglich zu halten, um Probleme durch Abhängigkeiten zu vermeiden. Dadurch sind Module untereinander lose gekoppelt und können unabhängig voneinander entwickelt, getestet und deployed werden [Richards 2015, 22].

Die Plugins können über verschiedene Wege mit dem Kern verbunden werden. Eine genaue Spezifikation zum Verbinden der Plugins mit dem Kern gibt es aber laut Architekturschema nicht, diese Entscheidung ist dem Entwickler überlassen und entsprechend der Anforderungen und Anwendungsumgebung zu treffen. Unabhängig von der Art der Verbindung definiert der Kern die Schnittstellen, um Plugins anzubinden. Diese Verbindung könnte dann beispielsweise über Web-Services, Messaging oder am einfachsten über direkte Objekt-Instanziierung innerhalb der gleichen Anwendung stattfinden [Richards 2015, 22-23].

Zwischen Plugins und Kern werden Verträge definiert, die die Kommunikation zwischen den beiden Komponenten regeln. Diese Verträge können in Form von Interfaces, Klassen oder auch Datenstrukturen definiert werden. Alle Plugins, müssen sich zwingend an die definierten Verträge halten, um mit dem Kern kommunizieren zu können. Alternativ können auch Adapter verwendet werden, um bestehende Plugins an den Kern und die Verträge anzupassen, wodurch wiederrum die lose Kopplung der Komponenten verbessert wird.

Durch diesen Aufbau ergibt sich jedoch das Problem, dass der Kern jederzeit über Verfügbarkeit und Erreichbarkeit der Plugins informiert sein muss. Um dieses Problem zu lösen, kann eine zentrale Plugin-Registry verwendet werden. Diese Registry enthält alle aktuell verfügbaren Plugins sowie die dazugehörigen relevanten Informationen wie zum Beispiel Name des Service, Verträge, Verbindungsdetails, etc. Der Kern der Anwendung kann dann zur Laufzeit auf diese Registry zugreifen und Plugins dynamisch laden [Richards 2015, 22].

Microkernel Architekturen können auch in andere Architekturmuster eingebettet werden, falls es nicht möglich sein sollte die gesamte Software in diesem Architekturmuster aufzubauen. Vor allem Teile von Anwendungen, die stark erweiterbar sein müssen, eignen sich gut für die Verwendung der Microkernel Architektur.

Ein klarer Vorteil dieses Architekturmusters ist die schnelle Reaktionsfähigkeit auf äußere Änderungen, da Anpassungen aufgrund der losen Kopplung größtenteils nur in den isolierten Modulen vorgenommen werden. Der Kern der Anwendung ist in den meisten Fällen schnell stabil und benötigt selten im Laufe der Entwicklung weitere Angleichungen. Geänderte Module können je nach Implementierung auch zur Laufzeit geladen oder hinzugefügt werden, was mögliche Downtime von bereits ausgelieferter Software minimiert [Richards 2015, 25].

Ein Beispiel dafür stellt die Entwicklung von Betriebssystem-Kernels dar, die auch namensgebend für dieses Architekturmuster ist. Deren Kern Komponenten sind in der Regel sehr stabil und implementieren vor allem grundlegende Funktionen wie Speicherverwaltung, Prozessverwaltung und I/O-Operationen. Weitere low-level Funktionalitäten wie beispielsweise Geräte Treiber oder Dateisysteme werden als Module in den Kernel geladen und können bei Bedarf hinzugefügt oder entfernt werden, was vor allem für die Unterstützung neuer Hardware wichtig ist.

Auch Entwicklungsumgebungen in der Softwareentwicklung nutzen oft Microkernel Architekturen, um die Support für verschiedene Programmiersprachen und Frameworks zu ermöglichen.

Zwar bietet das Beispiel der E-Commerce Anwendung keinen klassischen Anwendungsfall für Microkernel Architekturen, jedoch kann die Verwendung von Microkernel Architekturen in Teilen der Anwendung trotzdem sinnvoll sein. Sowohl Zahlungs- als auch Versandfunktionalitäten könnten, wie in Abbildung 3 dargestellt, in Module ausgelagert werden, um die Anwendung durch weitere Dienstleister erweitern zu können.

Logik zum Verarbeiten der Zahlungen und Versandinformationen wird dann in den Modulen implementiert, die auf den Kern der Anwendung aufbauen. Dabei ist jedoch ein Großteil der Business-Logik im Kern der Anwendung enthalten, was nicht dem eigentlichen Gedanken der Microkernel Architektur entspricht.

Abgesehen davon erhöhen Microkernel Architekturen inhärent die Testbarkeit der Software, da Module nur lose Kopplung gekoppelt sind. Jedes Modul kann unabhängig voneinander getestet werden und fehlende Module durch Stubs ersetzt werden, wodurch sich

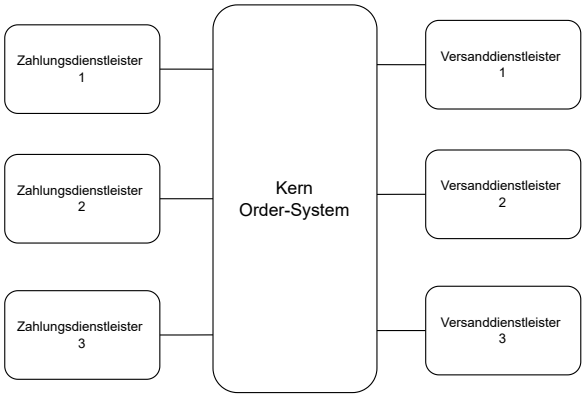

Abb. 3. Aufbau der E-Commerce Anwendung mit Microkernel Architektur

während der Entwicklung auf einzelne Module isoliert konzentriert werden kann [Richards 2015, 26]. Weiterhin können Verhaltensweisen von anderen Modulen durch Mocks simuliert werden, um Testzustände zu erzeugen und das Verhalten der Anwendung zu verifizieren. In Agilen Umgebungen, in denen das Testen von Software eine wichtige Rolle spielt, ist die Verwendung von Microkernel Architekturen daher besonders sinnvoll.

Eine Herausforderung bei der Verwendung von Microkernel Architekturen kann jedoch der Entwurf der Kern-Komponente darstellen. Da alle anderen Module auf den Kern aufbauen, muss dieser sehr sorgfältig und stabil entwickelt werden, um die Funktionalität der gesamten Anwendung zu gewährleisten. Diese Rolle sollten vor allem erfahrene Entwickler übernehmen, da sich Design-Fehler der Kern-Komponente oder Verträge negativ auf die Entwicklung der Module auswirken können. Sollte der Kern der Anwendung angepasst werden, so müssen tendenziell auch alle Module überprüft oder aktualisiert werden, was zu erheblich erhöhten Entwicklungszeiten führen kann und zuvor gewonnene Vorteile der Microkernel Architektur zunichte macht.

Aufgrund der initial hohen Komplexität, die mit der Entwicklung des Kerns einhergeht, stellt die Microkernel Architektur nicht die beste Wahl dar, wenn es darum geht schnell eine erste Version der Software auf den Markt zu bekommen. Die wesentlichen Vorteile, die die Microkernel Architektur bietet, zeigen sich erst im späteren Verlauf der Entwicklung, wenn die Anwendung erweitert werden muss. Sowohl Iterationen als auch Auslieferungszeiten sind dann sehr kurz, da Module unabhängig voneinander entwickelt werden und die Anwendung schnell an neue Anforderungen angepasst werden kann. Diese Vorteile können die Architektur in ausgewählten Anwendungsfällen sehr geeignet für agile Entwicklungsumgebungen machen.

4.3 Event-Driven Architecture

Die Event-Driven Architecture wählt als Basis einen anderen Ausgangspunkt als die bisherigen Architekturmuster. Während bei letzteren Komponenten Dienste bereitstellen, welche von anderen Komponenten explizit genutzt werden, verhalten sich Dienstbereitstellende Komponenten in der Event-Driven Architecture reaktiv, werden also implizit von Dienst-konsumierenden Komponenten genutzt [Garlan and Shaw 1994]. Ein System reagiert somit asynchron auf Zustandsänderungen, also Ereignisse in diesem System [Manchana 2021]. Die in dieser Architektur minimalen Einheiten, welche Informationen einer Zustandsänderung kapseln, werden *Events* genannt. Die Idee der impliziten Behandlung von Ereignissen ist nicht neu und taucht erstmals 1994 im von Garlan und Shaw publizierten Papier *„An introduction to Software Architecture“* auf.

Betrachten wir im Folgenden die Basis-Bestandteile der Event-Driven Architecture:

- Ereignis (englisch *Event*): Kapselt Information einer Zustandsänderung eines Systems
- Produzent (englisch *Producer*): Komponente, die Event erzeugt
- Herausgeber (englisch *Publisher*): Komponente, die, von Produzenten erzeugte, Events publiziert
- Konsument (englisch *Consumer*): Komponente, die auf publizierte Events reagiert
- Vermittler (englisch *Mediator*): Komponente zwischen Produzenten und Konsumenten - filtert Events und verteilt diese auf Konsumenten
- Event-Bus: Oft auch *Event-Broker* genannt - bietet die Infrastruktur für die Gesamtheit der Vermittler

Abstrakt kann ein Event als Vertrag zwischen Produzenten und Konsumenten am Event-Bus betrachtet werden. Der Konsument nutzt die Spezifikation des Events am Bus, der Produzent implementiert jene Spezifikation. Abbildung 4 stellt diesen Vertrag dar.

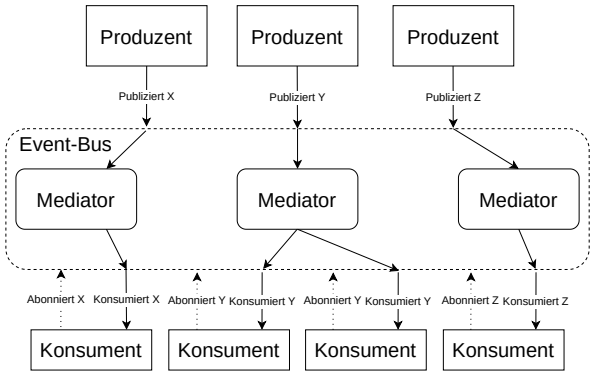

Abb. 4. Vertrag zwischen Produzenten und Konsumenten am Event-Bus

Durch den Vertrag weisen die Events am Event-Bus starke Kohäsion und somit lose Kopplung auf. Diese lose Kopplung minimiert nicht nur kaskadierende Fehler, sondern ermöglicht agilen Entwickler-Teams durch klar abgegrenzte Features einfach definierbare Iterationen - eine Menge von Events, deren Erzeugung und Konsumierung.

Weiter sind Events oft nah an dem, was Ereignisse in realen Prozessen sind, also domain-driven. Gebündelt ermöglichen obige

Punkte die kontinuierliche Auslieferung von Software in kurzen Intervallen.

Außerdem garantiert die asynchrone Behandlung von Ereignissen zusammen mit der loosen Kopplung maximale Skalierung. Daher sind Event-Driven Architekturen besonders für datenintensive Echtzeit-Anwendungen wie IoT (Internet of Things) und Analytics geeignet [Siddiqui et al. 2023].

Betrachten wir erneut das E-Commerce-Beispiel aus der Einleitung. Dafür definieren wir drei Arten von Events:

- `OrderCreated`: Ein Event, das genau dann erzeugt wird, wenn eine neue Bestellung aufgegeben wird
- `PaymentProcessed`: Ein Event, das genau dann erzeugt wird, wenn der Bezahlvorgang abgeschlossen wird
- `ShipmentInitiated`: Ein Event, das genau dann erzeugt wird, wenn die Bestellung versandt wird

Weiter teilen wir die Funktionalität ähnlich wie bei der Microservice-Architektur in die drei verschiedenen Dienste `OrderService`, `PaymentService` und `ShipmentService` auf.

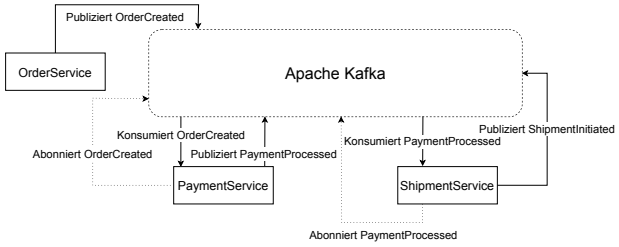

Abb. 5. E-Commerce-Beispiel mit Event-Driven Architecture

Wie Abbildung 5 zeigt, sind alle drei Dienste Produzenten und Publisher, erzeugen also Events und veröffentlichen diese. Die Dienste `PaymentService` und `ShipmentService` sind zudem Konsumenten, sodass ersterer auf Events des Typs `OrderCreated` und zweiterer auf Events des Typs `ShipmentInitiated` reagiert. Eine beispielhafte Implementierung des `PaymentService` mit Apache Kafka als Event-Broker ist im Anhang A.1 zu finden. Die vollständige Implementierung des E-Commerce-Beispiels ist bei GitHub [1] zu finden.

Das Beispiel zeigt, dass die Event-Driven Architektur mit weiteren agilen Strukturen wie Microservices kombiniert werden kann, was die Agilität der Architektur weiter erhöht. Die damit einhergehende Komplexität stellt teilweise hohe Anforderungen an die Entwickler. Aufgrund der Asynchronität der Behandlung von Ereignissen ist die Testung des Systems meist schwer und die Fehlerbehandlung essentiell. Mögliche Problemquellen schließen dabei unter anderem Event-Verlust, erhöhte Latenz und Inkonsistenz ein. Die hohen Anforderungen an die Entwickler verlangen viel Vertrauen in jene, einer der zentralen Punkte des agilen Manifests [Michl 2018]. Insgesamt weist die Event-Driven Architecture also eine sehr hohe Agilität auf und ist damit besonders für moderne Software und ihre stetig wechselnden Anforderungen geeignet.

[1] https://github.com/Beleg-6-EAP/demo-eda-ecommerce

5 FALLSTUDIEN UND PRAXISBEISPIELE

Blah …

6 DISKUSSION

7 ZUSAMMENFASSUNG UND AUSBLICK

LITERATUR

David Garlan and Mary Shaw. 1994. *An Introduction to Software Architecture.* Technical Report CMU/SEI-94-TR-021. https://insights.sei.cmu.edu/library/an-introduction-to-software-architecture/ Accessed: 2025-Jan-2.

Ramakrishna Manchana. 2021. Event-Driven Architecture: Building Responsive and Scalable Systems for Modern Industries. *International Journal of Science and Research (IJSR)* 10 (01 2021), 1706–1716. https://doi.org/10.21275/SR24820051042

Thomas Michl. 2018. *Das agile Manifest – eine Einführung.* Springer Berlin Heidelberg, Berlin, Heidelberg, 3–13. https://doi.org/10.1007/978-3-662-57699-1_1

Mark Richards. 2015. *Software Architecture Patterns.* O'Reilly Media, Inc. value pages.

Hassaan Siddiqui, Ferhat Khendek, and Maria Toeroe. 2023. Microservices based architectures for IoT systems - State-of-the-art review. *Internet of Things* 23 (2023), 100854. https://doi.org/10.1016/j.iot.2023.100854

A CODE-BEISPIELE

A.1 Event-Driven-Architecture

```java
package demo.eda.service;

import demo.eda.event.OrderCreatedEvent;
import demo.eda.event.PaymentProcessedEvent;
import demo.eda.model.Payment;
import demo.eda.repository.PaymentRepository;
import lombok.RequiredArgsConstructor;
import org.springframework.kafka.annotation.KafkaListener;
import org.springframework.kafka.core.reactive.ReactiveKafkaProducerTemplate;
import org.springframework.stereotype.Service;
import reactor.core.publisher.Flux;
import reactor.kafka.receiver.KafkaReceiver;
import reactor.kafka.receiver.ReceiverOptions;

import java.util.UUID;

@Service
@RequiredArgsConstructor
public class PaymentService {

    private final PaymentRepository paymentRepository;
    private final ReactiveKafkaProducerTemplate<String, PaymentProcessedEvent> producer;

    @KafkaListener(topics = OrderCreatedEvent.TOPIC, groupId = "payment-service")
    public Flux<Void> processPayment(ReceiverOptions<String, OrderCreatedEvent> receiverOptions) {
        return KafkaReceiver.create(receiverOptions)
                .receive()
                .flatMap(record -> {
                    final OrderCreatedEvent event = record.value();
                    final Payment payment = new Payment(UUID.randomUUID().toString(), event.getOrderId(), true);
                    return paymentRepository.save(payment).flatMap(savedPayment -> {
                        final PaymentProcessedEvent paymentEvent =
                            new PaymentProcessedEvent(
                                    savedPayment.getOrderId(),
                                    savedPayment.getId(),
                                    savedPayment.isSuccess());
                        return producer.send(PaymentProcessedEvent.TOPIC, paymentEvent).then();
                    });
                });
    }
}
```

Listing 1. Service-Implementierung des `PaymentService` in Java Spring Boot 3.4.1 mit Apache Kafka als Event-Broker

B ÜBUNGSAUFGABEN

B.1 Übungsaufgabe 1

Blah …